Make Halting Problem an Unsolved Problem Again by Carefully Designing a Purely Functional Programming Language

tsao chi

a certain High School :)

tsao-chi@the-lingo.org

Abstract

While the halting problem is proven to be undecidable in conventional purely functional programming languages, it's possible to design a Turing-complete purely functional programming language such that the problem of implementing an interpreter which will finish running for any input program is unsolved by breaking some fundamental properties of conventional purely functional programming languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Halting Problem; Language Design;

1. Introduction

The key is to cancel the uniqueness property of normal forms, then the proof of the undecidability of whether the process of interpreting an expression will halt, known as Turing's proof [2], is invalid.

1.1 The Functional Programming Language

It's a Scheme dialect. Mostly the same as Scheme, except for the following items.

1. purely functional and lazy
2. the Exception Handling System

2. Background - How the idea was conceived

When I was in junior high school, I was playing with my Scheme-like purely functional language and came up with these two features.

2.1 the Exception Handling System - Exceptions are just ordinary values

Functions that do not expect an exception as input will generate a new exception which usually (it's not enforced by language specialization, so "usually") includes the original exception when the input is an exception.

2.2 the cancellation of the uniqueness property of normal forms

I observed that there are some expressions which have not naturally unique normal form, such as converting a finite set into a list.

3. the bottom-rewriting rule

Just announcing the cancellation of uniqueness cannot truly cancel uniqueness, because it can still be proven. Therefore, some rules are needed to cancel the uniqueness of what we care about.

With these two features, it's reasonable to allow an interpreter to interpret an expression that will run forever as an exception. I call it the bottom-rewriting rule.

4. Effects of the bottom-rewriting rule

4.1 Non-uniqueness caused by the bottom-rewriting rule

Consider this example. Interpret `a` or `b` or the entire expression as an exception is all reasonable.

```
(letrec ([a (car b)] [b (car a)]) a)
```

4.2 Rewrite Range

Interpreting the entire program as an Exception conforms to the bottom-rewriting rule, but it will bypass the program's exception handling system. So an interpreter should try to avoid this situation.

5. How is this language Turing-complete?

A computational system that can compute every Turing-computable function is called Turing-complete (or Turing-powerful). [1]

The bottom-rewriting rule does nothing to Turing-computable function, so this language is Turing-complete.

6. Further work

1. Solve the problem.
2. Formalize this language.

References

[1] Turing completeness (wikipedia). 2020. URL `https://web.archive.org/web/20200707074705/https://en.wikipedia.org/wiki/Turing_completeness`.

[2] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society, 2 (published 1937)*, 1936.